KinectFusion based on Vulkan

Jinjiang You (jinjiany)
Yixin Fei (yixinf)
Peipei Zhong (peipeiz)

April 5, 2024

1 Topic and Motivation

Accurate and real-time camera localization and scene reconstruction are crucial for mixed reality, robotics, and autonomous vehicles. RGB-D SLAM techniques stand out, especially when compared to more sensor-intensive SLAM approaches, because they leverage cost-effective and compact sensors. The interest in RGB-D SLAM systems surged following the release of Microsoft's Kinect in 2010. RGB-D sensors, which combine a standard RGB camera with a depth sensor, enable the direct acquisition of depth information with reasonable accuracy in real time using affordable hardware.

In this project, we aim to implement KinectFusion[5] based on Vulkan[3] and test it on various datasets to demonstrate its real-time performance. Because of the cross-platform and low-level nature of Vulkan, we can utilize GPU to accelerate the computation on both PC and mobile devices. Because Vulkan is essentially a graphics API, we can also integrate KinectFusion with other graphics rendering pipelines to add AR features. Our objective is to enhance the accessibility of 3D mapping technology, making it lightweight and user-friendly and facilitating a more engaging and immersive user experience.

2 Related Topics

2.1 RGB-D SLAM

The introduction of the Kinect by Microsoft has significantly heightened interest in SLAM systems that leverage RGB-D data. These sensors combine a standard RGB camera with a depth sensor, offering a distinct advantage over monocular and binocular cameras. RGB-D cameras can directly measure pixel depth using infrared structured light and/or Time of Flight (TOF) technologies, eliminating the need for complex computations. TOF technology operates on a simple yet effective principle: it calculates distances by timing the laser's journey. Consequently, SLAM algorithms based on RGB-D data offer an alternative to traditional visual and visual-inertial SLAM methods, capable of generating detailed, dense maps for a more accurate representation of the environment. This advantage is especially evident in environments with low texture, where the depth sensor's input ensures more reliable mapping and localization.

The KinectFusion algorithm[5] a pioneering real-time RGB-D sensor-based method, employs a four-step process: measurement, pose estimation, reconstruction update, and surface prediction. Following its introduction, several RGB-D based SLAM algorithms have emerged. For example, SLAM++[6] is an object-oriented SLAM algorithm that leverages the repeatability of objects and structures within a known scene to enhance efficiency and scene characterization through loop closure detection. The dense visual odometry SLAM (DVO-SLAM) algorithm[2] relies on a keyframe-based approach, minimizing the photometric error between keyframes to determine depth values, pixel coordinates, and camera motion. The RGBDSLAMv2[1] algorithm sets itself apart within the RGB-D framework by concentrating on feature extraction. It implements the RANSAC algorithm to deduce the transformations between matched features and uses the Iterative Closest Point (ICP) algorithm for refined pose estimation.

Given KinectFusion's seminal role in the development of RGB-D based SLAM, this project aims to implement a real-time RGB-D volumetric fusion system inspired by KinectFusion. Our goal is to adapt this technology for broader accessibility, such as mobile devices, thereby advancing the application of SLAM technologies in various settings.

2.2 Vulkan

Vulkan is a cross-platform low-level graphics API. Different from other graphics APIs (e.g. OpenGL, Direct3D), Vulkan allows the developers to have low-level control over the graphics rendering process, thus leading to better performance and less surprising driver behavior compared to existing APIs. Besides, Vulkan has the advantage of being fully cross-platform and allows developers to develop for Windows, Linux, MacOS[4], and Android at the same time.

Besides traditional graphics rendering, Vulkan also supports compute shaders. This opens up the world of general-purpose computing on graphics processor units (GPGPU). GPGPU means that developers can do general computations on the GPU, something that has traditionally been a domain of CPUs. With GPUs becoming more and more powerful and more flexible, many workloads that would require the general-purpose capabilities of a CPU can now be done on the GPU in real time.

As a very low-level API, Vulkan has the shortcoming that developers have to work with a significantly more verbose API. Generally, it is easier to develop graphics applications using OpenGL or develop parallel computing applications using CUDA. However, performance matters in our project, and we want our SLAM system to be cross-platform. OpenGL cannot help us achieve the best performance and CUDA is only available on NVIDIA's cards. So we will stick with Vulkan to develop our project. So far, there are a lot of open-source KinectFusion implementations (e.g. CUDA-based), but no Vulkan-based implementation. Our project will definitely contribute to the open-source community.

3 Implementation Design

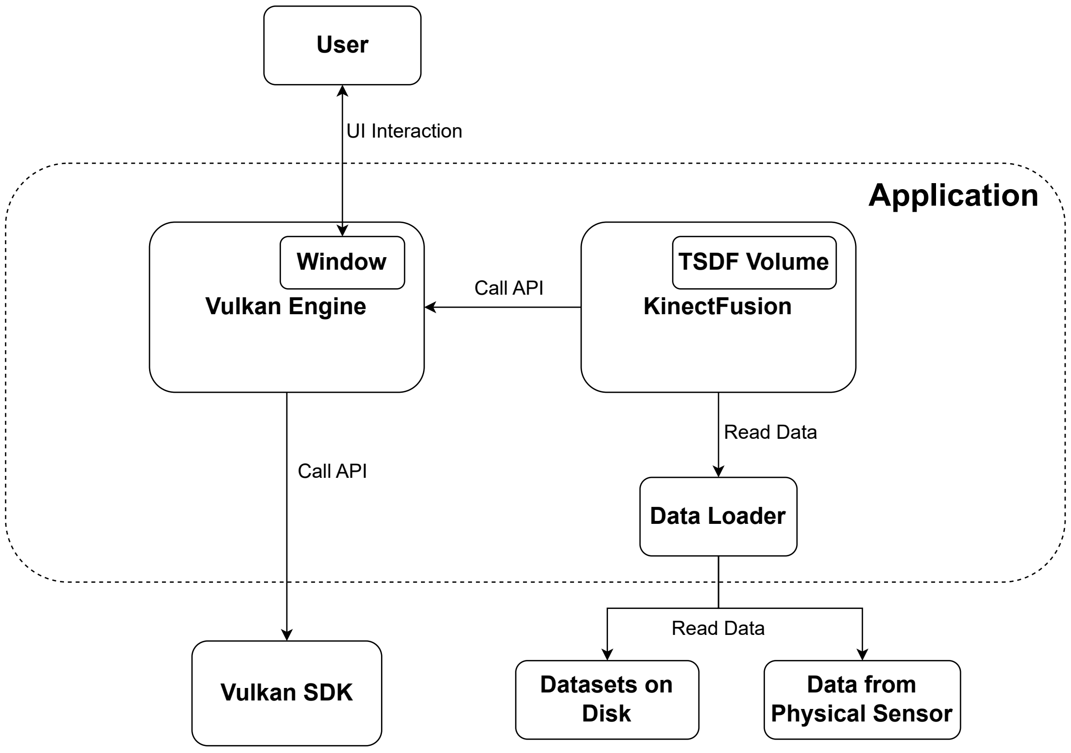

Figure 1: Overall pipeline for our implementation.

The overall pipeline is shown in fig. 1. Our system is divided into 4 modules.

3.1 Vulkan Engine

We implemented a *Vulkan Engine* class to decouple most of the Vulkan API interaction from other modules. The engine will initialize the Vulkan context including creating the Vulkan instance, logical device and queues, memory allocator, the swapchain, and the descriptor pool. It guarantees that 3 queues are created for graphics, compute, and transfer, respectively, so that other modules can use these queues to perform different tasks to achieve the best performance. Besides, the engine also creates several graphics pipelines for 2D/3D simple/lambertian rendering. They are useful for visualizing the camera trajectory, the reconstructed scene, or other AR objects.

The engine class does not create compute pipelines. It only exposes the compute queue and computes command buffers for other modules to use. It is the other modules' responsibility to load compute shaders and create compute pipelines.

3.2 TSDF Volume

KinectFusion[5] uses the TSDF (Truncated Signed Distance Function) volume to represent the global model, effectively creating a dynamic and continuous 3D representation of the environment in real-time. The volumetric approach is key to how KinectFusion smoothly combines all the depth data from the Kinect sensor as it moves around. We assign each voxel in the volume a value that represents its distance to the nearest surface boundary, with a sign indicating whether it's inside or outside the surface, and then KinectFusion can efficiently track changes and update the global model with high precision. Far away from any surface, the values are "truncated" or cut off to save on processing power and memory, which is the meaning of "truncated" in TSDF.

The strength of using a TSDF volume lies in its ability to handle the inherent noise and incompleteness of depth data from the Kinect sensor. Rather than treating each incoming frame of depth data as an isolated snapshot, KinectFusion accumulates evidence over time, leading to a more robust and detailed 3D model.

Moreover, the use of a TSDF volume helps real-time camera tracking by enabling the system to quickly find correspondences between the live depth data and the current global model. This is important for maintaining the alignment of the 3D model as the camera moves, ensuring that the reconstructed environment remains consistent and stable over time.

In our implementation, the TSDF volume is abstracted into a separate class called *TSDFVolume*. Currently, it is a dense volume, but it is possible to extend it to support voxel hashing or color reconstruction. The class creates a storage buffer to store the voxels, and the corresponding descriptor set layout and the descriptor set to pass the storage buffer to shaders. Besides, the class also creates a pipeline to initialize the volume.

3.3 Fusion Algorithm

The core algorithms of KinectFusion[5] include four components, and the overall system workflow is shown in fig. 2.

1. **Surface measurement**: We obtain the raw depth measurements from the Kinect device, and then generated a dense vertex map and normal map pyramid.
2. **Surface reconstruction update**: In the global scene fusion process, once the position and orientation (pose) are figured out by tracking the depth data from a new camera shot, the measurements of surfaces are added into the 3D model of the scene. This model is kept in a special 3D grid format known as a truncated signed distance function (TSDF).
3. **Surface prediction**: Instead of estimating the camera's position from one frame to the next, as done in other studies, KinectFusion improves accuracy by comparing the latest depth data directly with the entire 3D model we've built so far. We do this by projecting our 3D model onto the new camera view to predict what the surfaces should look like, and then we match this prediction with the actual new depth data we just got.
4. **Sensor pose estimation**: We achieve live sensor tracking by using a multi-scale ICP alignment between the predicted surface and current sensor measurement.

In our implementation, all the core algorithms of the fusion are implemented in the *KinectFusion* class. It provides the following API:

- Given a compute graphics camera model (e.g. a perspective camera), perform ray casting in the volume to generate a color image and a depth image for rendering.
- Given a computer vision camera model (e.g. a pinhole camera), perform ray casting in the volume to generate a point cloud for ICP.

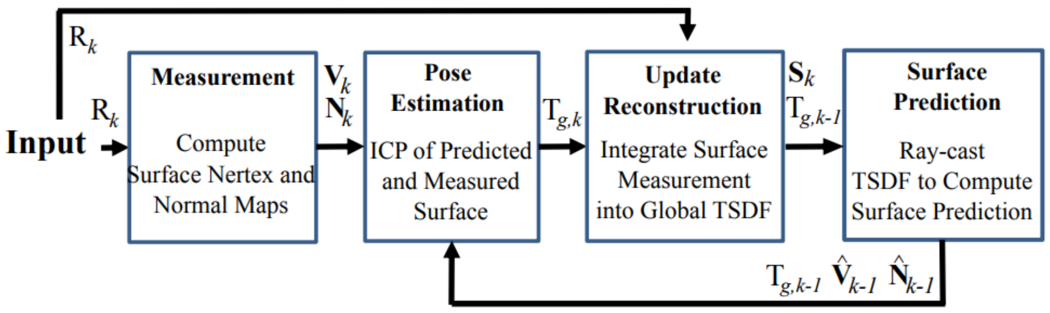
Input
R_k
R_k
Measurement
Compute Surface Nertex and Normal Maps
V_k
N_k
Pose Estimation
ICP of Predicted and Measured Surface
$T_{g,k}$
Update Reconstruction
Integrate Surface Measurement into Global TSDF
S_k
$T_{g,k-1}$
Surface Prediction
Ray-cast TSDF to Compute Surface Prediction
$T_{g,k-1}$ $\hat{V}_{k-1}$ $\hat{N}_{k-1}$

Figure 2: The overall workflow of the core fusion algorithm.

- Given a new frame, preprocess it including bilateral filtering and building a pyramid of vertex maps and normal maps.
- Given a preprocessed new frame, perform frame-to-model tracking to compute the frame's pose.
- Given a new frame and its pose, register it into the global model.

3.4 Data Loader

To support various kinds of test data, we abstract the data loading part into a separate virtual base class called *DataLoader*.

The data loader provides the following API:

- Get the frame size (width and height).
- Get the camera parameters (focal length, principal points).
- Get the next frame's data.

To test our implementation on existing datasets, we can implement derived classes that load data from the disk. They are also called offline data loaders.

It is also possible to implement a derived class that wraps around a physical sensor. In this way, the data is streamed from the sensor and the fusion system processes the data in real-time.

The basic idea of this design is that the fusion system is not aware of the data source. It runs as if the data is streamed from a true physical sensor.

4 Progress

We have finished implementing the Vulkan Engine and TSDF volume modules. We have finished implementing the ray casting algorithm in the Fusion module. For more information, keep track of our repository.

5 Evaluation Methods

Since we decide to implement the fusion algorithm on both PC and mobile devices, there are two ways to evaluate our implementation. On PC, we will test on some famous SLAM datasets like the TUM RGB-D [7] dataset. We intend to use the automatic evaluation tool in the TUM RGB-D to evaluate the quality of the resulting maps. Additionally, we will analyze the algorithm's frame rate to demonstrate its capability for real-time operation. On IOS devices, we will use their depth cameras to capture real-world data and perform a fusion algorithm on a background thread to show that it can run in real-time on Mobile devices.

References

[1] F. Endres, J. Hess, J. Sturm, D. Cremers, and W. Burgard. 3-d mapping with an rgb-d camera. *IEEE transactions on robotics*, 30(1):177–187, 2013.

[2] C. Kerl, J. Sturm, and D. Cremers. Dense visual slam for rgb-d cameras. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2100–2106, 2013. doi: 10.1109/IROS.2013.6696650.

[3] Khronos. Vulkan Cross platform 3D Graphics, . URL https://www.vulkan.org/.

[4] Khronos. MoltenVK Run Vulkan on iOS and OS X, . URL https://moltengl.com/moltenvk/.

[5] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136, 2011. doi: 10.1109/ISMAR.2011.6092378.

[6] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. Kelly, and A. J. Davison. Slam++: Simultaneous localisation and mapping at the level of objects. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1352–1359, 2013. doi: 10.1109/CVPR.2013.178.

[7] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.